Assignment 4: Sequential models

We'll continue to use high-level machine learning APIs to explore what we can do with recurrent neural networks, building a contextual tagger and making our first attempt at the lemmatization task. I'll continue to give implementation notes for Keras users, although as before, you can use whatever API you would like.

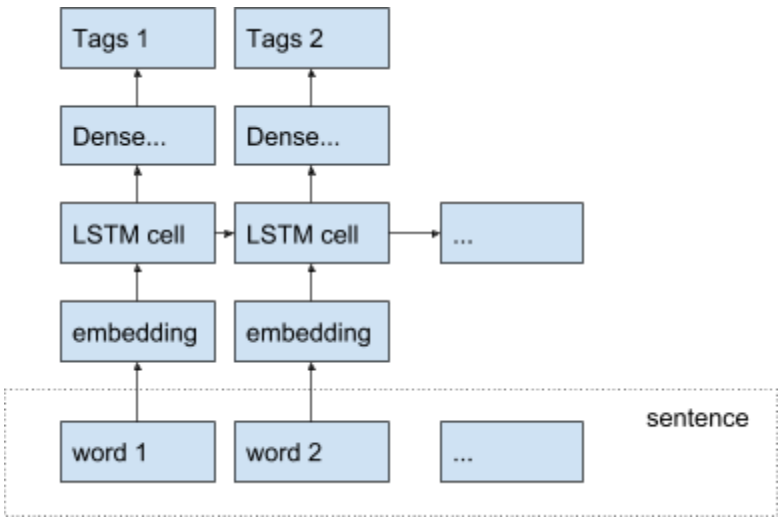
As usual, the model definitions below cut some corners in the interests of speed and simplicity---I've tried to point out (or ask you to point out) some of the most glaring omissions, but please let me know if there are other things you think would make them perform better.

You should write a nicely-formatted lab report (in Word or PDF format) explaining what you did and what happened. Explain any decisions you made which were not pre-specified in the assignment.

Part 1: sequence tagger

This part of the assignment will focus on the part-of-speech tagging task which we've been working on all along. We've discussed the idea that the context in which a word occurs can inform us about its tag dimensions; the LSTM network will allow us to incorporate this information in a relatively straightforward way.

The basic network we will construct has the following structure:



Some things to notice about this structure:

- It is *not* an encoder-decoder (seq2seq) architecture. The outputs line up 1-to-1 with the inputs.

- Everything above the “LSTM cell” unit in the graph is conceptually the same as the networks in the previous assignment.
- The input is sentence-by-sentence, not word-by-word.

Input representations: For ease of implementation, we’ll treat the length of each sentence as a constant. Compute this constant by taking the 95th percentile sentence length in the training data. (One way to do this is with **numpy.percentile**). We will pad any sentence shorter than this with dummy words, and truncate any sentence longer than this (just refusing to tag the extra words). Keras is entirely capable of tagging the full-length sentences, but this requires you to write your own batch generator and you already did that in Assignment 2, so I won’t make you do it again.

We’ll also treat the vocabulary size as a constant. In this case, we’ll use the 10000 (ten thousand) most common words in the training data, plus two special tokens: an “unknown word” token which we use for everything else, and a “padding” token which we use to fill up sentences shorter than the maximum length.

This allows us to create a fixed-dimension input matrix. There are two ways to encode this input:

- A matrix of *(n examples) x (max sentence length) x (10000)*, in which entry *i, j, k* = 1 when word *i* of sentence *j* is vocabulary item *k*. This is a so-called **one-hot** encoding. In this case, it’ll take a lot of memory, so if your preferred API forces you to go with it, you may want to generate your batches incrementally after all.
- A matrix of *(n examples) x (max sentence length)*, in which entry *i,j* gives an integer code corresponding to word *i* of sentence *j*. If you use Keras, this is the easiest option.

Remember to convert all words which are not in the vocabulary to the **unknown** token, and to add copies of the **pad** token to fill up every sentence to a constant number of words.

In either case, the outputs should be one-hot: *(n examples) x (max sentence length) x (n tag dimensions)* and have 1 where sentence *i*, word *j* has tag *k*.

Model design: You can attach an **Embedding** layer above the input matrix. Use a 64-dimensional embedding space. (This was acceptable for my dev experiments in English, and as usual, I’m trying to avoid murdering your computer.)

If you wanted to, you could map these embeddings directly to outputs by using **TimeDistributed(Dense(...))**, which would create a **Dense** layer, then attach a copy of that layer above every embedding in the input. This architecture would have no recurrent connections (no sideways arrows on the diagram), so it would basically be a very complicated version of the baseline in Assignment 1, except that for some reason it only knew 10000 words. **You are not required to build this system**, although it can be a useful sanity check if you are having trouble doing the next part.

Add an **LSTM** layer above the embedding, again with 64 hidden units. Use **return_sequences=True** so that you get a state at every timestep.

Add **TimeDistributed(Dense(num_tags, activation="sigmoid"))** to create a **Dense** layer and attach a copy of that layer above each LSTM output. **Optional: you may try out other layers** in between the LSTM and the outputs, including batch normalization, more dense layers, more LSTMs or the Bidirectional wrapper in this part of the model.

As usual, use **binary crossentropy** as your loss. You will probably want to run more than 1 epoch (pass over the data). I used Adam and got reasonable performance on English in 3 or 4. (You can use the *sample_weight* argument to mask out losses for padding characters. In my development runs, this didn't matter--- predicting "no tags" for the padding character is just too trivial to make any difference.)

Run the system on the standard set of languages from the previous assignment and report the standard metrics. Your scoring implementation should properly account for the system's failure to tag words that are cut off by the truncation. Do not score any tags predicted for padding (although a properly trained system won't produce any).

Analysis questions:

We expect the LSTM to be able to exploit context. Let's see if it does this. On the English_EWT dataset, find three words which appear >10 times in the dev data, and which occur with more than one tag vector, and for which the majority tag vector accounts for less than 70% of the occurrences. Show 10 occurrences of each one in context; give the predicted tags and the correct tags. Does the LSTM appear to learn to use context?

For which languages is the input representation used here suitable? For which is it ineffective and how would you fix it?

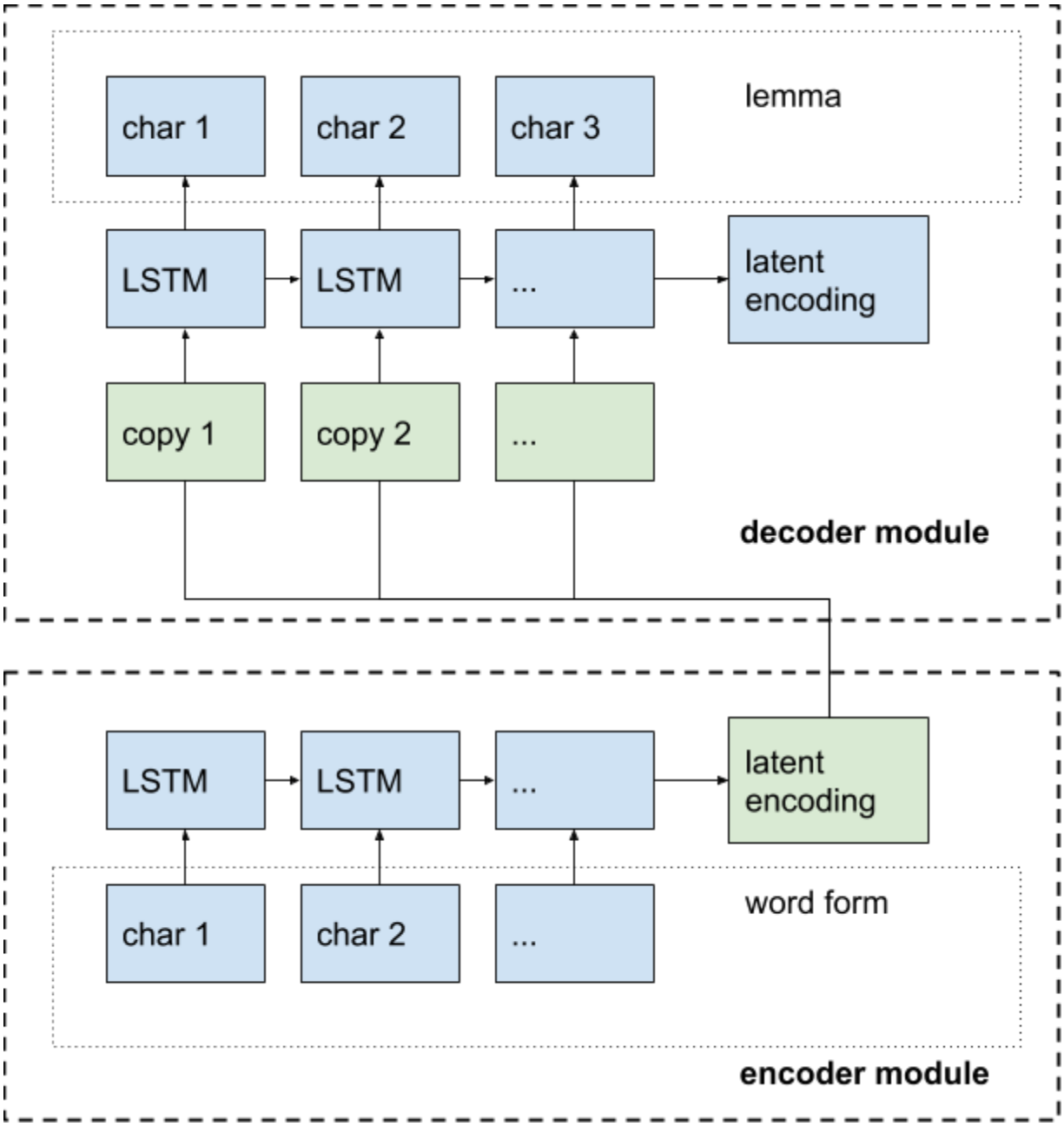
Part 2: sequence-to-sequence lemmatizer

In this part of the assignment, we'll work on the lemmatization task. As a reminder of the task definition, look back at the SigMorphon example:

```
# sent_id = newsgroup-groups.google.com_eHolistic_417b09016150421f_ENG_20051120_175800-0002
# text = One Huge Opportunity.
1       One     one     _       _       NUM     _       _       _       _
2       Huge    huge    _       _       ADJ     _       _       _       _
3       Opportunity     opportunity     _       _       N;SG    _       _       _       _
4       .       .       _       _       _       _       _       _       _
```

In this task, we'll be mapping column 2 (the **form**) to column 3 (the **lemma**). In the example shown here, this is simply downcasing, but in general it also involves stripping away a lot of other inflectional material.

We'll use a simple sequence-to-sequence architecture which is closely based on this Keras example (https://keras.io/examples/addition_rnn/) and which looks like this:



Input representations: Again, we'll treat both input and output sequences as fixed-length, to avoid having to write our own batch generator. Compute these constants by taking the 95th

percentile word form length and lemma length in the training data. Assign each character in the training data a unique index number. Represent both the input and the output using one-hot matrices; the input is *(n examples) x (max word form length) x (n chars + 1)* and the output is *(n examples) x (max lemma length) x (n chars + 1)*, where the extra character is the padding value.

Model design: You can assemble this using **LSTM**, **RepeatVector** and **Dense** layers. Use the "softmax" activation for the output layer and categorical crossentropy loss, since the characters are mutually exclusive. Look at the example code in the link above--- most of this can be copied. **Optionally,** you can mess around with other model extensions.

Evaluation: You can decode your model predictions using the following procedure:

- Create a dictionary which maps **index -> character** by inverting your **character -> index** dictionary.
- Each row of the prediction matrix represents a word.
- Use the **argmax** function to find the maximum value in each column (ie, the index of the most probable character).
- Assemble all these characters into a string.
- Truncate this string at the first occurrence of the pad character.

Compute the 0/1 accuracy for predicted lemmas. (You get a point if your prediction matches the true lemma and otherwise you get nothing.)

Baseline: Compare your accuracies to the following simple baseline: predict every word form as its own lemma.

Model tuning: Try out your lemmatizer on English (EWT):

- Training for 10 epochs. Use 32 LSTM states. Report the loss, the accuracy, and the predictions for 5 lemmatized words after every epoch.
- Repeat the experiment with 64 LSTM states.

Analysis questions:

What does the lemmatizer learn to do quickly/easily?

What does it have a hard time with?

Run the same set of languages as the previous assignment. Are there patterns for where it works well/poorly?

An obvious improvement to the lemmatizer would be to use **attention**. Explain why this would help and for which cases you expect it to be most effective.